Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets, © JOT 2011
Online at http://www.jot.fm.

Efficient Retrieval and Ranking of Undesired Package Cycles in Large Software Systems

Jannik Laval[a] Jean-Rémy Falleri[a] Philippe Vismara[b]
Stéphane Ducasse[c]

a. Univ. Bordeaux, LaBRI, UMR 5800, F-33400 Talence, France
http://sphere.labri.fr

b. LIRMM, UMR5506 CNRS - Université Montpellier 2, France
http://www.lirmm.fr/
MISTEA, UMR729 Montpellier SupAgro - INRA, Montpellier, France

c. RMoD, Inria Lille Nord Europe, France
http://rmod.lille.inria.fr

Abstract Many design guidelines state that a software system architecture should avoid cycles between its packages. Yet such cycles appear again and again in many programs. We believe that the existing approaches for cycle detection are too coarse to assist developers to remove cycles from their programs. In this paper, we describe an efficient algorithm that performs a fine-grained analysis of cycles among application packages. In addition, we define multiple metrics to rank cycles by their level of *undesirability*, prioritizing cycles that are the more undesired by developers. We compare these multiple ranking metrics on four large and mature software systems in Java and Smalltalk.

Keywords software architecture; software re-engineering; package cycle; package dependency.

1 Introduction

Large object-oriented software projects are usually structured in *packages* (or *modules*). A package is primarily used to group together related classes which define a functionality of the system. Classes belonging to the same package should be built, tested, versioned, and released together. Martin consequently proposed to see the package as the software release unit [Mar02]. Considering a package as a unit, a package dependency is a dependency representing multiple class dependencies from a package to another. There are multiple kinds of class dependency. We consider in

this paper three kinds: inheritance, interface and references, presented in details in Sect. 3.2.2.

Design guidelines state that cyclic dependencies between packages should be avoided [Par78, Mar02]. Indeed, packages depending cyclically on each other are to be understood, tested, released, or deployed together. Sometimes, dependencies appear during the development process and form cycles between packages that do not need to evolve together. We name them *undesired* cycles because they make the program less modular. In contrast, some packages evolve and are released together, they could be involved in a cycle that is *desired*. We explain in detail the concept of package cycle in Sect. 2.2.

Several tools and approaches have been developed over the years [Vai04, MT07b, SJSJ05, LDDB09] to help the developers to detect and/or remove cycles. Yet, an exhaustive experimental study [MT07a] shows that in a lot of programs, classes are involved in huge cyclic dependencies. It seems therefore plausible that the way cycles are detected is not sufficient to help developers to address them.

We claim that the existing approaches have two main issues. First, some focus on cycles between classes, when cyclic dependencies at the package level should have the priority. Indeed classes are not deployment units, and a lot of cycles among classes are due to the associations, being thus totally expected. Second, and most important, existing approaches are all based on the same algorithm by Tarjan [Tar72]. This algorithm finds the maximum sets of packages depending (directly or indirectly) on each other, called *strongly connected components* (SCC) in graph theory. Within a SCC, a package is in cycle with all other packages, and there can be multiple cycles in one SCC. In our experience, we have seen software systems with a single huge SCC containing dozens of packages. For example, ArgoUML has a SCC containing 38 packages. The above algorithm becomes useless in such cases as it does not provide further information to understand and remove the cycles.

A dependent problem which is not well addressed in current approaches is ranking cycles so that the most "undesired" ones are given top priority for removal. Indeed, not all cyclic dependencies have the same importance. In a system of packages where a hierarchy can be extracted from the naming conventions (as in Java), a cycle between *ui* and *core* packages should be avoided as it hampers reuse and deployment of the system. On the contrary, a package such as *ui.internal* can be in cyclic dependency with *ui* without serious consequences, since they both implement the same functionality. We further discuss these issues as well as the prevalence of package cycles in four Java programs in Sect. 2.

Our approach advocates the decomposition of a SCC in multiple *short cycles* covering all dependencies of the SCC. Computed *short cycles* usually involves two to four packages. They are therefore easy to understand and to remove, if necessary. Developers can iterate over a set of short cycles and assess them one by one rather than dealing with the single large set of packages contained in the SCC. Moreover, our approach is able to rank the extracted cycles, *prioritizing the ones that are the more undesired*.

In this paper, we present two major contributions to assist developers in understanding and removing cyclic dependencies in software systems:

- First, we present an efficient algorithm that decomposes a SCC. This algorithm retrieves a set of short cycles that covers all dependencies of the SCC. It has a polynomial time and space complexity (Sect. 3.1).
- Second, we introduce new metrics that measure the level of undesirability of a

cycle. The diameter metric needs a package containment tree, the weight-based metric is based on the number of class dependencies, and the diameter-weight-based metric is an average of the two previous metrics. We evaluate them on real software systems.

We expand on our previous work [FDL+11] in two ways. First, we propose multiple ranking metrics: the diameter metric was already presented in our previous work, the two new metrics: weight-based metric and diameter-weight-based metrics are presented in Sect. 3.2.2 and Sect. 3.2.3 respectively. Second, our approach is validated against four large and mature programs in Java (Sect. 4).

2 Motivation

This section presents a small study showing why the SCCs are not fine-grained enough to assist developers in understanding and removing cycles in large programs (Sect. 2.1). Then, it explains using an example why some cycles among packages can be desired by their developers (Sect. 2.2).

2.1 Limitation of the main cycle detection algorithm

Most of the approaches perform cycle detection by using an algorithm [Tar72] that is capable of finding the *maximum sets of packages that depend (directly or indirectly) on each others*. Such a set of packages is called, in the graph theory, a *strongly connected component* (SCC). In a SCC, each package is in cycle with all other packages, and cycles exist only among the packages of a same SCC. To remove package cycles, it is therefore necessary to remove several dependencies among the packages of a given SCC. We believe that the SCCs are not fine-grained enough to help the developer to understand and remove the undesired dependencies in their programs. Indeed, they indicate *which* packages are involved in cyclic dependencies, but they can not explain *how*. Whenever a SCC contains only a few packages, it remains possible to visualize the dependencies between them and to remove the cycles. On the other hand, when a SCC contains a lot of packages, it does not help the developer at all. Indeed, if it contains dozens of packages, it becomes hard to understand how packages connect with each other to create the SCC.

To show that mature and large programs can contain huge SCCs, we performed a small experiment. We selected four mature and medium-sized Java programs: ArgoUML[1], JEdit[2], Choco[3] and AntLR[4]. On these programs we computed: #**P** the number of packages, #**LSCC** the size of the largest SCC and **LSCCR**: the ratio of packages in the largest SCC.

Tab. 1 shows that the programs we selected contain large SCCs. In ArgoUML the largest SCC contains almost half of the packages (see the **LSSCR** measure). Worse, in JEdit almost two thirds of the packages are in the largest SCC, whereas the total number of packages is not too large. Apart from AntLR, the size of the largest SCC in the programs of our corpus will make their understanding hard (see the #**LSCC** measure).

[1] http://argouml.tigris.org
[2] http://www.jedit.org
[3] http://www.emn.fr/z-info/choco-solver
[4] http://www.antlr.org

Program	#P	#LSCC	LSCCR
ArgoUML 0.28.1	79	38	48%
JEdit 4.3.1	29	18	62%
Choco 2.1.0	147	38	26%
AntLR 3.2	31	7	23%

Table 1 – Measures among packages and package cycles on the Java programs. **#P** is the number of packages, **#LSCC** the size of the largest SCC and **LSCCR** the ratio of packages in the largest SCC.

2.2 Desired and undesired cycles

In the introduction, we stated that not every cycle should be removed. In fact, we believe that a significant proportion of the cycles among packages are desired by the developers. To show this, let us take the example of the JFace[5] main widget library used in the Eclipse development environment. A large effort has been devoted to its design by several software design experts. We therefore assume that the cycles present in JFace are not accidental. Package *jface.text* is dedicated to the text widgets. This package provides classes such as *TextViewer*. Package *jface.text.hyperlink* is dedicated to the management of textual hyperlinks. In JFace, there is a cycle between *jface.text* and *jface.text.hyperlink*. The *TextViewer* class is able to display texts containing hyperlinks and therefore *jface.text* depends on *jface.text.hyperlink*. Also, *jface.text.hyperlink* uses a lot of classes and interfaces defined in *jface.text*. For instance a hyperlink is able to trigger text events and therefore depends on the *TextEvent* class, which is defined in the *jface.text* package. Therefore *jface.text.hyperlink* depends on *jface.text*. In this case, the complexity of the hyperlink motivates its isolation in package *jface.text.hyperlink*. Yet it is not necessary to break the cycle with *jface.text* as it would make no sense to release one without the other.

More generally, in several languages such as Java, a package can contain other packages, leading to a package containment tree. It is usual that when a package is too large (i.e., contains too many classes), it is split in several sub-packages. In this case it is very likely that cycles exist between these sub-packages. These sub-packages are highly coupled, which does not represent a poor design but is necessary for readability [TSWW11].

3 Our Approach

In this section, we present our two contributions:

- First, we present an efficient algorithm that decomposes a SCC. This algorithm retrieves a set of short cycles that covers all dependencies of the SCC. It has a polynomial time and space complexity (Sect. 3.1).
- Second, in Sect. 3.2, we introduce new metrics that evaluate the level of undesirability of a cycle. Each metric is based on a characteristic of the package architecture. One metric, called *diameter*, is based upon the distance between

[5] http://wiki.Eclipse.org/index.php/JFace

packages involved in the cycle (Sect. 3.2.1). A second metric is based on the number of class dependencies between packages involved in the cycle (Sect. 3.2.2). A third metric is computed based on the two previous metrics (Sect. 3.2.3).

3.1 A new cycle retrieval algorithm

3.1.1 Intuition of our algorithm

To explain better the intuition of our new algorithm, let us first introduce a sample class diagram, shown in Fig. 1. From this class diagram, we extract the directed graph shown in Fig. 1. This graph shows the dependencies between the packages, therefore we call it a *package dependency graph*. On this graph, the SCCs are rounded by dashed circles.

In the previous section, we stated that the algorithm that computes SCCs is not fine-grained enough to help the developers to understand and remove cycles from their programs. Fortunately, another algorithm from the graph-theory literature is able to perform a fine-grained analysis of cycles in a directed graph [Tar73]. It computes the set of *elementary cycles*. A cycle is elementary if no node (here no package) appears *more than once* when enumerating the sequence of nodes in the cycle. For instance, in our sample graph of Fig. 1, this algorithm finds the six elementary cycles shown in Fig. 1. Figuring out if an elementary cycle should be removed or not is straightforward, it only requires to decide if the dependencies involved in the cycle are correct. The fact that the elementary cycle is short eases this decision. Unfortunately, the number of elementary cycles in a directed graph can be exponential. For instance the algorithm of [Tar73] spent one hour on each program presented in Tab. 1 without terminating. Therefore, this algorithm does not scale on programs composed of many packages.

We introduce a new algorithm that still computes elementary cycles in a SCC but that retrieves only a polynomial number of them, reducing time and space complexity. Indeed, some elementary cycles can be seen as redundant. In Fig. 1, cycle $\mathcal{C}_5$ is not useful if we consider cycles $\mathcal{C}_1$ and $\mathcal{C}_4$. Indeed, the dependencies covered by $\mathcal{C}_5$ have already been covered by the two other cycles, $\mathcal{C}_5$ contains more dependencies than $\mathcal{C}_1$ or $\mathcal{C}_4$. We reduce the number of cycles by selecting only a subset of the elementary cycles, ensuring that each dependency of the SCC is covered by at least a cycle. Still, to get all dependencies covered in Fig. 1, it is possible to select cycles $\mathcal{C}_2$, $\mathcal{C}_3$, $\mathcal{C}_6$, and either $\mathcal{C}_1$ and $\mathcal{C}_4$, or the cycle with more dependencies $\mathcal{C}_5$. We assume that the more dependencies a cycle contains, the harder it is to understand, because it requires the analysis of many dependencies. Therefore our final solution is to select for each dependency *one of the shortest cycles going through the dependency.*

3.1.2 Mathematical model

A package dependency graph G is a couple (P, E) with P a set of nodes (the packages) and E a set of edges (dependencies between the packages). An edge is a couple $(s, t) \in P^2$ where s is the source and t the target package. There is an edge from a package s to a package t iff a class of s uses a class of t. We denote a path in G by a sequence of nodes, written this way: (a, b, c), where every node has an edge to its successor. We denote a cycle by such a sequence of nodes: $x \to y \to z$, the last node being implicitly linked to the first one.

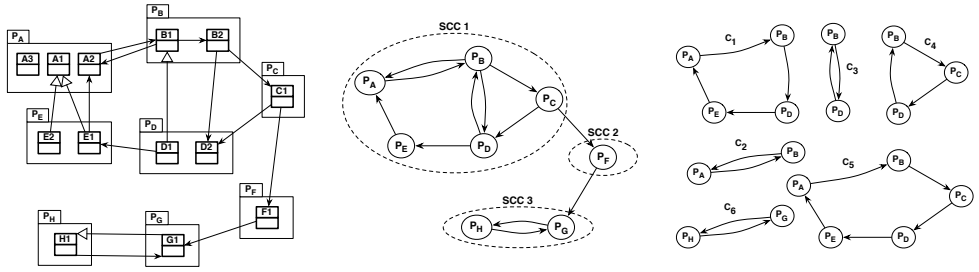

Figure 1 – A sample class diagram (left), the corresponding package dependency graph (midde, the dashed lines round the SCC) and the elementary cycles for this graph(right).

3.1.3 Details of our algorithm

To understand the algorithm, it is important to notice that cycles exist *only* among the nodes of the same SCC. Also, the set of SCCs of a directed graph is a partition of its nodes. Therefore as a preliminary step to our algorithm, we retrieve the SCCs from the directed graph using the algorithm of [Tar72], remove the inter-SCCs edges, then run our algorithm on each SCC containing more than two nodes (SCCs of size one cannot contain a cycle). The SCCs of size two contain only one cycle involving the two nodes. Therefore on the graph of Fig. 1, only *SCC 1* is considered by our algorithm, while *SCC 2* is discarded and *SCC 3* directly leads to the creation of the cycle $p_G \rightarrow p_H$. In the following, we therefore focus on what happens in a SCC of size greater than two. To find shortest cycles, we use the well-known *breadth-first search* (BFS) algorithm. This algorithm can be used to find the shortest path between two nodes in a graph where the edges are unweight. A SCC has the following property: for each possible pair of nodes x, y of the SCC, there is a path from x to y and from y to x. A simple algorithm to find a shortest cycle for every edge of a strongly connected graph is therefore to perform for each edge $(x, y) \in E$ a BFS from target node y going back to source node x. Indeed since there is an edge from x to y, this edge is already the shortest path from x to y. Since we are in a SCC, it is mandatory that at least a path exists from y to x. A shortest path from y to x (found by the BFS) concatenated with the edge (x, y) would therefore be a shortest cycle in which this edge is involved.

The only problem of this simple algorithm is that it requires a BFS for each edge of the graph. Since there are fewer nodes than edges in a strongly connected graph, it would be better to perform a BFS only for each node of the graph. The idea is therefore to gather the parents $A = \{y \in P | (y, x) \in \Gamma_P^-(x)\}$ of a node x, and perform a BFS from x until all its parents $y \in A$ are found. This way, only one BFS is performed for each node. The pseudo code of this optimized version is given in Algorithm 1. To avoid the retrieval of identical cycles, we consider that two cycles are equals if the first is a cyclic permutation of the second. For instance $c \rightarrow a \rightarrow b = a \rightarrow b \rightarrow c$. To have a fixed order to represent the cycles and compare them efficiently, we always place the lowest node (using the lexicographic order) at the beginning of the cycle. We call this operation *normalize*. For instance $\mathit{normalize}(c \rightarrow a \rightarrow b) = a \rightarrow b \rightarrow c$.

Let us see how this algorithm works on *SCC 1*, shown in Fig. 1. Remember that the edge from p_C to p_F has been deleted because it is an inter-SCCs edge. The set of nodes is $P = \{p_A, p_B, p_C, p_D, p_E\}$. We start with an empty set of cycles: $\mathcal{C} = \{\}$. Here are the steps followed by our algorithm:

Algorithm 1: Our cycle retrieval algorithm

```
Data: A strongly connected package dependencies graph G = (P, E)
Result: A set of shortest cycles C
begin
    C ← {} ;                                              // the set of cycles
    for x ∈ P do
        V ← {} ;                                  // the set of the visited nodes
        A ← {z ∈ P|(z, x) ∈ E} ;                      // the set of the x parents
        x.bfs_parent ← ∅ ;                        // the path followed by the BFS
        Q ← (x) ;                                  // a queue, initialized with x
        /* BFS from x that stops when all parents of x are found  */
        while size(A) > 0 do
            p ← pop(Q);                       // removes the first element of Q
            B ← {z ∈ P|(p, z) ∈ E} ;              // the set of the p children
            for y ∈ B do
                /* if y has not been visited or put on the stack    */
                if y ∉ V ∪ Q then
                    y.bfs_parent ← p;
                    push(Q, y) ;                        // adds y at the end of Q
                /* if a parent of x is reached                      */
                if y ∈ A then
                    c ← () ;              // the list of the nodes of the cycle
                    i ← y;
                    /* builds the cycle                             */
                    while i ≠ ∅ do
                        add(c, i);
                        i ← i.bfs_parent;
                    /* adds the cycle to the set of cycles          */
                    normalize(c);
                    if c ∉ C then C ← C ∪ {c};
                    remove(A, y) ;                          // removes y from A
            V ← V ∪ {p} ;                                   // p is now visited
```

1. The first node being picked up is p_A. Therefore, $A = \{p_E\}$. The BFS starting from p_A will find p_E by the following path: (p_A, p_B, p_D, p_E). Since $\mathcal{C}$ is empty, the cycle $\mathcal{C}_1 = p_A \rightarrow p_B \rightarrow p_D \rightarrow p_E$ is added to $\mathcal{C}$. $\mathcal{C} = \{\mathcal{C}_1\}$.
2. The second node being picked up is p_B. $A = \{p_A, p_D\}$.
   - The BFS started from p_B will find p_A by the following path: (p_B, p_A). This cycle is normalized to $\mathcal{C}_2 = p_A \rightarrow p_B$ and added to $\mathcal{C}$. $\mathcal{C} = \{\mathcal{C}_1, \mathcal{C}_2\}$.
   - The BFS started from p_B will find p_D by the following path: (p_B, p_D). The cycle $\mathcal{C}_3 = p_B \rightarrow p_D$ is added to $\mathcal{C}$. $\mathcal{C} = \{\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3\}$.
3. The third node being picked up is p_C. $A = \{p_B\}$. The BFS starting from p_C will find p_B by the following path: (p_C, p_D, p_B). After normalization, it becomes $\mathcal{C}_4 = p_B \rightarrow p_C \rightarrow p_D$ and it is added to $\mathcal{C}$. $\mathcal{C} = \{\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3, \mathcal{C}_4\}$.
4. The fourth node being picked up is p_D. $A = \{p_B, p_C\}$.
   - The BFS started from p_D will find p_B by the following path: (p_D, p_B). This cycle is normalized to $\mathcal{C}_3$ and therefore is not added to $\mathcal{C}$. $\mathcal{C} = \{\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3, \mathcal{C}_4\}$.
   - The BFS started from p_D will find p_C by the following path: (p_D, p_B, p_C). This cycle is normalized to $\mathcal{C}_4$ and therefore is not added to $\mathcal{C}$. $\mathcal{C} = \{\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3, \mathcal{C}_4\}$.
5. The fifth and last node picked-up is p_E. $A = \{p_D\}$. The BFS starting from p_E will find p_D by the following path: (p_E, p_A, p_B, p_D). This cycle is normalized to $\mathcal{C}_1$ and therefore is not added to $\mathcal{C}$. $\mathcal{C} = \{\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3, \mathcal{C}_4\}$.

Finally, we have $\mathcal{C} = \{\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3, \mathcal{C}_4\}$. We can notice that in contrast to the enumeration of all elementary cycles (see Fig. 1), the cycle $p_A \rightarrow p_B \rightarrow p_C \rightarrow p_D \rightarrow p_E$ is not retrieved by our algorithm. This is expected since our algorithm, as previously explained, select for each dependency of a SCC one of the cycle with the lesser number of dependencies going through it.

3.1.4 Complexity of Algorithm 1

Let $n = |P|$ be the number of nodes and $m = |E|$ be the number of edges. In the worst case, we pick-up a different cycle for every edge, the maximum number of cycles is therefore m. We split the computation of the worst-case time complexity in three parts: most time spent in the pre-processing step (finding the SCCs), most time spent in the BFSes, and most time spent to add the cycles in the cycle set. Since we work with strongly connected graphs, we have $m \geq n$.

1. The worst case time complexity of the algorithm that computes the SCCs in the pre-processing step is $O(n + m)$ [Tar72].
2. The worst case time complexity of a BFS in a graph is $O(n + m)$. Since we perform a BFS for every node of the graph, it leads to a $O(n(m+n))$ complexity for the BFSes.
3. The addition of a cycle in the set of cycles can be done in $O(n \times log(n))$ using appropriate data structures (like a self-balancing binary search tree). In the worst case, we try to add the same cycle involving all packages for each edge. Therefore the worst case time complexity for the additions is $O(m \times n \times log(n))$.

Since $m \geq n$, the overall complexity of our algorithm is $O(m \times n \times log(n))$. Since the number of packages in a program cannot be too large (we consider $1,000$ packages as a fair upper-bound), this complexity is perfectly acceptable to be applied at development-time (for an immediate feedback) as well as maintenance-time (for an in-depth architecture assessment). This analysis is confirmed by the running times measured on the programs evaluated in Tab. 4.

3.2 Undesired cycles ranking metrics

In the previous section, we showed how we efficiently retrieve cycles from a package dependency graph. Unfortunately, there can be many cycles, especially in a large and complex program. A developer is not going to inspect manually all the cycles, because it is a tedious and time-consuming task. Moreover, a significant amount of these cycles is probably *desired*, like we have seen in Sect. 2. To assist in understanding and removing the cycles, it is critical to propose the cycles that are the most *undesired*.

This is the purpose of the three following hypotheses and ranking metrics presented in the following subsections.

- First, we assume that if a cycle involves packages that are far away in the package containment tree, it is very plausible that the cycle is *undesired*. We compute the diameter metric based on this hypothesis (Sect. 3.2.1). Note that this metric was introduced in our previous work [FDL+11].
- Second, we consider that if a dependency between two packages is caused by only few dependencies between their contained classes, it is very plausible that the dependency has been introduced accidentally. We propose the weight metric based upon this assumption in Sect. 3.2.2.
- At last, the combination of the two previous metrics might provide a more fine-grained ranking. We propose this metric in Sect. 3.2.3.

To order the cycles, the following rules are applied to each ranking metric:

- Each ranking metric is a function $R : \mathcal{C} \rightarrow \mathbb{R}^+$ where $\mathcal{C}$ is the set of shortest cycles, retrieved by the algorithm presented in Sect. 3.1.
- The cycles are ranked from highest to lowest value of R, except for the weight-based metric (Sect. 3.2.2) ranked from lowest to highest.
- If two cycles have an equal ranking value, the number of packages contained in the cycle is used to rank the cycles (the fewer packages it has, the higher it is ranked). If two cycles have an equal ranking value and number of packages, they are ranked using the lexicographic order.

In the rest of the section, we detail our metrics.

3.2.1 Diameter metric

To define the *diameter* metric, we assume that packages are named using a convention defining a *containment tree*. This is the case in many languages such as Java, C#, Ruby, or PHP.

To illustrate the phenomenon described in Sect. 2, let us imagine the sample package containment tree shown in Fig. 2. In this package tree, let us imagine a cycle

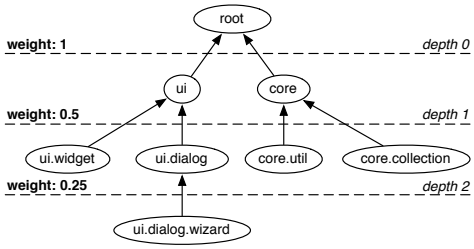

Figure 2 – A sample package containment tree, with the value associated to the edges.

between *ui.dialog.wizard* and *ui*. It is common that a class in a package uses classes of its parent packages. It is also possible that in the parent package, several classes depend on the classes of the sub-packages (such as factory classes). In our example, *ui.dialog.wizard* is likely to use several classes defined in *ui*, like the class *Widget*. It is also likely that *ui* furnishes a factory class to create wizards (such as *WizardFactory*), that uses the different wizards defined in *ui.dialog.wizard*. In this case this cycle would be totally desired since the developer would neither use nor deploy *ui* without *ui.dialog.wizard*. In contrast, let us imagine a cycle between *core* and *ui*. Although the dependence from *ui* to *core* is normal, it is unlikely that a package such as *core* requires *ui* to be used or deployed. The cycle is strongly undesired.

To order the cycles, we use the package containment tree to define a distance between two packages: for instance the number of edges required to go from a package to the other package. We assume that the further away are the packages involved in a cycle, the more undesired the cycle is. Unfortunately, with this definition of distance, the packages *ui.dialog.wizard* and *ui* are at the same distance from each other as *core* and *ui* (two edges). To deal with this problem we add a second assumption: the farther away the common ancestor between two packages is from the root of the tree, the less the distance between them is significant. For instance, the common ancestor between *ui.dialog.wizard* and *ui* is *ui*, while the common ancestor between *core* and *ui* is *root*.

To deal with the two previously described assumptions, we define a function that assigns a high value to the edges close to the root and a low value to the edges far from the root. The value of an edge depends on its depth. For an edge e at depth d, the value $w(e) = \frac{1}{2^d}$. In Fig. 2 we can see the weight associated to edges. For example the two edges $ui \rightarrow root$ and $core \rightarrow root$ have a weight equals to 1. The distance between two packages $\delta : P^2 \rightarrow \mathbb{R}^+$ is then equal to the sum of the values of the edges that lead in the shortest path from the first package to the second one. For instance $\delta(core, ui) = 2$, $\delta(ui.widget, ui.dialog) = 1$ and $\delta(ui.dialog.wizard, ui) = 0.75$.

We can now define our metric that indicates the level of undesirability of a cycle, called *diameter* (denoted by D). It is defined as the worst possible distance between two packages contained in the cycle. More formally, let C be a cycle, and let $P(C)$ be the set of packages contained in the cycle, $D(C) = max(\{\delta(x, y)|x, y \in P(C), x \neq y\})$. Let us imagine that there is the following cycle: $ui \rightarrow ui.widget \rightarrow core$. The diameter of this cycle is $D(ui \rightarrow ui.widget \rightarrow core) = 2.5$ because $\delta(core, ui) = 2$, $\delta(ui, ui.widget) = 0.5$ and $\delta(core, ui.widget) = 2.5$. We also have: $D(ui \rightarrow ui.widget \rightarrow ui.dialog) = 1$ because $\delta(ui, ui.widget) = 0.5$, $\delta(ui, ui.dialog) = 0.5$ and $\delta(ui.dialog, ui.widget) = 1$.

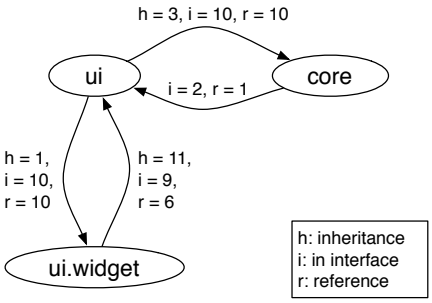

Figure 3 – A sample package architecture, with the weight associated to the edges.

3.2.2 Weight-based metric

It is common that multiple packages are highly dependent on each other, because they represent the same module in the software architecture. For instance, the study of Taube-Schock el al. [TSWW11] shows that a high coupling does not always represent a poor design, and some high coupling might be necessary for a good design. In this case, cycles between highly coupled packages are desired. To avoid suggesting removal of these cycles, we will search for cycles involving packages that have the fewest dependencies between them, using a ranking metric called *weight*.

To define the *weight* metric, we assume that a package dependency is a collection of class dependencies. We consider three kind of class dependencies to compute our metric. *Inheritance* (h) represents an inheritance relationship between two classes. *Interface* (i) between classes a and b means that b is publicly available in a (for instance by being the return type of a public method). This kind of dependency is originally defined in [MT07a]. Finally we call *reference* any other kind of dependencies between two classes (such as method invocation or attribute access). Let us take the sample package architecture with class dependencies shown in Fig. 3. In this package graph, a cycle between *ui.widget* and *ui* is desired. In our example, the dependencies *ui* $\rightarrow$ *ui.widget* and *ui.widget* $\rightarrow$ *ui* have high weights (respectively, 21 and 26 class dependencies). In this case, the packages are highly coupled and the cycle *desired*. The cycle between *core* and *ui* has a dependency with a weight equal to 3, which is 7 times lower than the other dependencies in the example. This cycle is undesired, the *core* package should not depend on the *ui* package.

To order the cycles, we define a weighting function $\omega : P^2 \rightarrow \mathbb{R}^+$ that assigns a high score to the edges representing lots of class dependencies and a low score to the edges representing few class dependencies. The score of an edge depends on the number of class dependencies. For an edge e from a package p_A to a package p_B with x_h inheritances, x_i interfaces and x_r references, weighted respectively with w_h, w_i, and w_r, the score between p_A and p_B is the aggregation $\omega(p_A, p_B) = w_h.x_h + w_i.x_i + w_r.x_r$. For any packages without class dependencies p_x and p_y, we set $\omega(p_x, p_y) = \infty$.

We can now define a metric based on this formula, denoted by W_{eq}, with $w_h = w_i = w_r = 1$. It is defined as the smallest possible score between two packages contained in the cycle. More formally, let C be a cycle, and let $P(C)$ be the set of packages contained in the cycle. $W_{eq}(C) = min(\{\omega(x, y) | x, y \in P(C), x \neq y\})$. In our example, $W_{eq}(core \rightarrow ui) = 3$, and $W_{eq}(ui \rightarrow ui.widget) = 21$. As previously explained, we assume that the smaller the score, the more undesired it is.

We also define another variation of this metric by assuming that the different

kinds of dependency have not the same importance. We consider indeed that an inheritance is a very strong dependency that has probably not been used accidentally. Interface is also a strong dependency. On the other, the odds that a reference is accidental are higher. Therefore, we define three weight coefficients to compute a weight metric that increases the weight, with a decreasing order of magnitude, of (in order): inheritances, interfaces and references. We call this metric W_{exp} and we choose $w_r = 10^1$, $w_i = 10^2$, and $w_h = 10^3$. We choose these values to clearly differentiate each kind of class dependency.

3.2.3 Diameter-Weight-based metric

The two previous metrics are based on the assumption that (i) the higher the diameter, the more the cycle is undesired, and (ii) the less the weight, the more the cycle is undesired. We assume that combining these two metrics provides a better ranking metric of undesired cycles. Based on the two previous metrics, we compute a new metric denoted by Z. It is defined as a combination of the worst possible distance and the smallest possible weight between two packages contained in the cycle.

Let us take the sample package structure shown in Fig. 3 in the context of Fig. 2. In this package graph, a cycle between *ui.widget* and *ui* is desired: the weights of edges are high related to the other edges and the distance value is minimal. The cycle between *core* and *ui* has a dependency with a weight equals to 3, which is a low value compared to the other edges, and the distance value is higher that the one between *ui.widget* and *ui*. This cycle is undesired.

Let C be a cycle, let $P(C)$ be the set of packages contained in the cycle. To combine δ and ω in a single metric, we normalize their values in the $[0, 1]$ interval. They are normalized as follows. $\delta_n(x, y) = \frac{\delta(x,y)}{\Delta}$ where Δ is the maximum value of $\delta(a, b)$ for any package a and b. Similarly, $\omega_n(x, y) = \frac{\omega(x,y)}{\Omega}$ where Ω is the maximal value of $\omega(a, b)$ for any edge (a, b) in E and $\omega_n(x, y) = 1$ if $(x, y) \notin E$. Finally, δ_n and ω_n are combined in $Z(C) = max(\{\frac{\delta_n(x,y)+(1-\omega_n(x,y))}{2} | x, y \in P(C), x \neq y\})$. For the computation of ω, we use the values $w_r = 10^1$, $w_i = 10^2$, and $w_h = 10^3$. We select these values because the results of W_{exp} are better than the ones of W_{eq} in our experiments (see Sect. 4).

4 Validation

We evaluate our approach on four large programs with an experiment involving their maintainers. Our approach can be used both at development-time and at maintenance-time. Nevertheless, we believe that it is harder to understand and remove a cycle at maintenance-time, because it is necessary to remember the past design decisions that led to its creation.

To show that our approach is useful we take the use-case where a developer uses our approach on his software at maintenance-time. We use the algorithm described in Sect. 3.1.3 to extract the cycles and rank them using each metric presented in Sect. 3.2.

4.1 Preparation of the data

We performed experiments on four different programs. These applications have enough package dependencies to be useful for our study: at least 20 packages and

50 package dependencies.

It can be noted that all these systems have cycles. This section presents the four studied systems and provides some useful metrics like number of packages and number of dependencies. All these metrics are available in Table 2.

RESYN-Assistant

RESYN-Assistant[6] is a Java program targeting the domain of organic chemistry. It includes several algorithms for perceiving molecular graphs according to their topological, functional and stereo-chemical features. The development of RESYN-Assistant started in 1996 at the LIRMM institute. The development team was composed of four persons: two researchers in computer-science, one PhD student in computer science and one PhD student in chemistry. Because of the turnover within the development team, and because it has mostly been developed by students having different research objectives, its architecture has decayed since the initial version. This system was already studied in our previous work. RESYN-Assistant has 313 classes distributed in 33 packages. There are 1886 class dependencies and 241 package dependencies.

Geco

Geco[7] is a Java desktop application to manage orienteering races, including registration of competitors, punch checking, and computation of results. It was developed over the course of three years by a single developer. Although it's a spare time project, some attention is given to the quality of the code and especially to the dependency between packages to maintain a clean architecture. Geco has 198 classes distributed in 23 packages. There are 1344 class dependencies and 93 package dependencies.

Kalimucho

Kalimucho[8] is a platform for dynamic QoS-driven deployment/reconfiguration of applications. It allows dynamic deployment and dynamic reconfiguration of applications on desktop computers, laptops and mobile devices with Java and/or Android, depending on the external context (bandwidth, lightness, energy, etc.). The development of Kalimucho started in 2009 at the LIUPPA institute. The Kalimucho project was accepted as a Mobile and Embedded project by Sun Microsystems, which offer two experimentation. The development team was composed of one PhD student and one researcher in computer science. The architecture has evolved since the initial version according to the different additions of new functionalities. Kalimucho has 118 classes distributed in 30 packages. There are 538 class dependencies and 130 package dependencies.

Herdsman

Herdsman is a tool for automating the analysis of the evolution of software ecosystems. The framework incorporates a database connection that allows us to get all relevant information obtained thanks to several mining tools, and provides a unified way for visualizing software communities evolution. Visualization tools are integrated to get a first quick overview of the evolution of different aspects of the software project under study. The tool is extensible to accommodate and different types of input and

[6] http://www.lirmm.fr/~vismara/resyn
[7] http://www.geco.webou.net/geco/index.html
[8] http://Java.net/projects/kalimucho

Characteristics	RESYN-Assistant	Geco	Kalimucho	Herdsman
Number of classes	313	198	118	242
Number of class dep.	1886	1344	538	1033
Number of packages	33	23	30	42
Number of package dep.	241	93	130	121

Table 2 – Characteristics of the studied system.

output, depending on the needs of the user. Herdsman has 242 classes distributed in 42 packages. There are 1033 class dependencies and 121 package dependencies.

4.2 Experiment

4.2.1 Dependency extraction

We extract the dependencies between classes from the Java byte-code of each program. To extract them, we use the Apache BCEL[9] library. With BCEL, we extract most of the dependencies between the classes, but some of them can be missed since it relies only on a static analysis of the code. The main issues are the following.

1. If a method is overridden, the dependency extracted using BCEL is always to the class that defines the method. But in reality the dependency could be to a subclass. For instance A, B and C are in three different packages, B is a subclass of A and B overrides a method M in A. An invocation of M in class C would be reported by BCEL as a dependency between C and A. However there could be a dependency between C and B, which would only be revealed by flow analysis of the code.

2. If a static member of a class is accessed without using a variable declaration, the dependency to the class can be overlooked by BCEL.

3. All dependencies caused by the use of the Java reflexion layer cannot be detected by BCEL. In each case, the dependencies extracted by BCEL are a subset of the real dependencies.

The first issue has no effect on our approach since method overriding will not prevent packages to be compiled, tested and deployed separately. The two last issues can affect our approach since packages using this kind of dependencies have to be compiled, tested and deployed together. As previously seen, some existing cycles are not detected by our tool, but every cycle detected by our tool is a real cycle. The amount of missed cycles depends on amount of use of the previously described mechanisms in the programs.

4.2.2 Experiment description

When using our approach to extract package cycles, one expects that the most undesired cycles will be ranked first and that the desired cycles will be ranked last. One also expects that our approach will extract short cycles, which are easier to understand than the long ones. To validate this, we set up the following experiment.

[9] http://jakarta.apache.org/bcel

For each program, we ask maintainers to evaluate all package dependencies with the three values *undesired*, *desired*, *unknown*. We provided a software called *DepsView*[10] to ease the extraction and the evaluation. A tutorial for the user is available on the wiki on DepsView's homepage. The software shows a list of all the package dependencies of the analyzed system. In addition, it provides a view that eases the understanding of the dependencies by showing the underlying class dependencies. The package dependency evaluation is independent of the cycle retrieval. This process provides a high objectivity of the developers about the cycles, because this software does not show if a dependency is in a cycle or not.

For each program, we compute and rank the cycles. First, we compute the distribution of the cycle sizes, to ensure that short cycles are retrieved. We then compare the maintainer answers to our algorithm results. To that extent, we count how many cycles in the k first ranked by the algorithms are undesired, and how many of the k last cycles are desired. We consider that an undesired cycle is a cycle with at least one *undesired* package dependency. A desired cycle is a cycle with no *undesired* dependency. Note that a desired cycle can contain *unknown* dependencies. In our study, the *unknown* dependencies are considered as not *undesired*

Using this information, we compute the precision over the k first cycles $FP_k = \frac{|\text{undesired}_k|}{k}$, where undesired_k is the number of undesired cycles in the top k cycles rated by the approach. In our experiment, we compute FP_{10}. This measure will show if our ranking metrics are able to rank highly undesired cycles. But it could be the case that there are only undesired cycles in the programs of our experiment. In this case, any ranking algorithm would have a good precision. To ensure the fact that our ranking metrics are able to rank low the desired cycles, we will also compute the precision over the k last ranked cycles $LP_k = \frac{|\text{desired}_k|}{k}$, where desired_k is the number of desired cycles in the low k cycles rated by the approach. In our experiment, we compute LP_{10}. If both FP_{10} and LP_{10} are close to 1, it means that our ranking metrics are useful.

We compute these values for all of the unwanted cycles taking metrics presented in Sect. 3.2. It provides four results for (i) the diameter ranking D, (ii) the weight ranking with equal weighting W_{eq}, (iii) the weight W_{exp}, and (iv) the diameter-weight ranking Z.

4.3 Results

4.3.1 Maintainers evaluation

The first step of our experiment was to ask maintainers to qualify package dependencies of their programs with the values *undesired*, *desired*, *unknown*. Tab. 3 provides the results of this evaluation. All the studied programs have undesired dependencies. Geco and Kalimucho have few undesired dependencies (*e.g.,* 6 and 4 respectively) which means that the package design is well structured. Resyn-Assistant and Herdsman have more undesired cycles. There is only one SCC in each software, and the minimal size is 10 for Geco. It means that it is necessary to decompose the SCC for a better understanding of the cycles.

4.3.2 Finding the cycles

Our algorithm finds 306 cycles in 9 milliseconds on RESYN-Assistant (mean time computed over 10 runs on a 2.2GHz Intel Core i7), 87 cycles in 3 milliseconds on

[10] http://popsycle.googlecode.com

Characteristics	RESYN-Assistant	Geco	Kalimucho	Herdsman
Undesired dependencies	17.02%	6.45%	3.08%	19%
Desired dependencies	80.08%	93.55%	93.08%	81%
Unknown dependencies	2.9%	0%	3.84%	0%
Number of SCC > 1	1	1	1	1
SCC max. size	29	10	17	20

Table 3 – Characteristics of the dependencies evaluated by maintainers.

Characteristics	RESYN-Assistant	Geco	Kalimucho	Herdsman
Number of short cycles	306	27	56	66
Number of undesired cycles	259	27	5	37
Time to find cycles (ms)	9	3	4	5

Table 4 – Results of our retrieval algorithm.

Geco, 121 cycles in 4 milliseconds on Kalimucho, and 98 cycles in 5 milliseconds on Herdsman. The time to compute cycles is low enough to be performed during the software development and integrated in the environment. The number of short cycles shows that a ranking metric is clearly needed to avoid to look at all cycles when re-engineering (Tab. 4).

The distribution of the cycle sizes is shown in Fig. 4. The largest cycles are of size 7, which is manageable. The majority of the cycles are of size 2, 3 or 4, which is totally suited for an easy understanding of the cycles. In comparison with the size of the unique SCC, the size of the cycles found by our algorithm is significantly smaller.

4.3.3 Precision using our different metrics

The program Geco is special because all the short cycles are undesired (see in Tab. 4). It means that the precision of FP_{10} takes the value 1.0, and the precision of LP_{10} takes the value 0.0. For the other systems, the best algorithm is the one with the higher precision. We compute the precision of each algorithm on the 10 first cycles and on the 10 last cycles of the provided list.

Fig. 5 and Fig. 6 show the precision over the 10 first and last cycles respectively. We see that the weight-based ranking metric W_{eq} does not provide good results. Precision of the diameter metric D is good for the four programs. It means that the first ranked cycles were, as expected, undesired. The majority of the last ranked cycles were, as expected, desired. Applied on Kalimucho, our metric does not show good results for FP_{10} measure. It is due to the low number of undesired cycles (5 on the 56). It is clearly difficult to find them. The precision of the weight-based metric W_{exp} is good but a bit lower than the precision of the metric D. This metric considers inheritance more important than the other dependencies by weighting it with 10^3. It means that package dependencies composed with inheritance are desired. The results of W_{exp} are better than results of W_{eq} for the four programs. It validates the postulate of the metric that inheritance is more desired than the other dependencies and would not be removed by engineers. Finally, the precision of the diameter-weight-based metric Z is good for the four programs. The FT measure is 1.0 for Herdsman, which is better

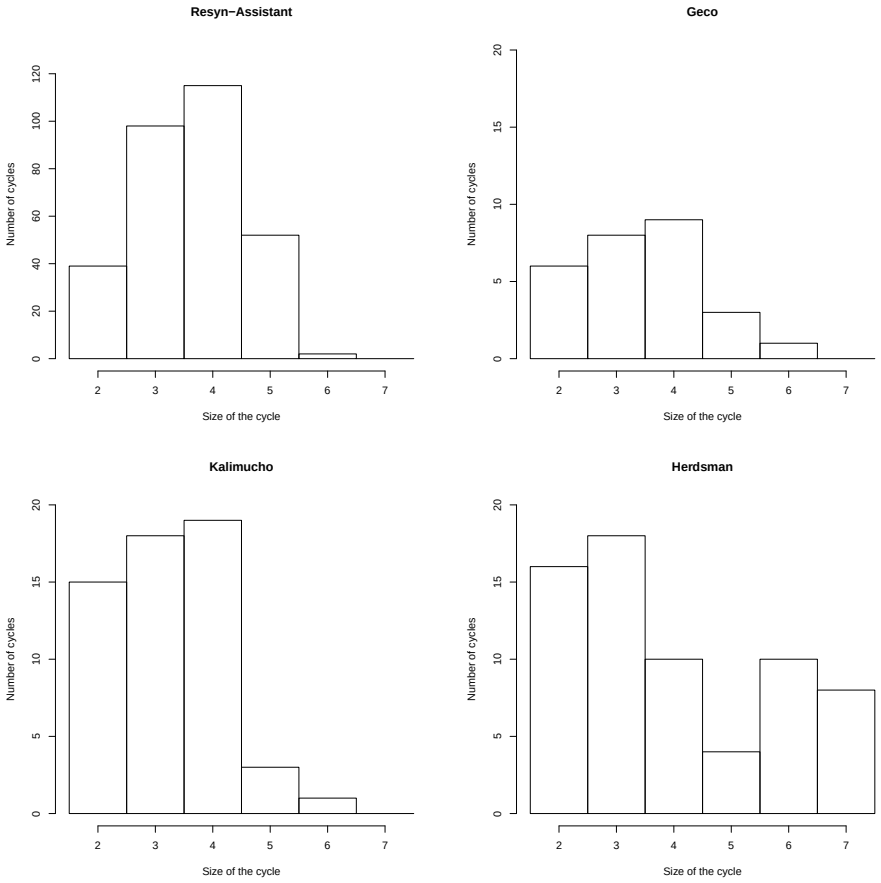

Figure 4 – The distribution of the cycle sizes in the four programs.

than the other metrics.

To conclude, the results show that ordering cycles by minimal weight (W_{eq} and W_{exp}) is not a good strategy. The diameter D provides good results, even if we compute an average with the weight (Z). It means that the package architecture in the studied programs depends on the containment tree.

The comparison of D and Z does not provide any conclusion because depending to the analyzed software, Z is more precise than D, less precise than D and equally precise as D. The conclusion is that Z is not superior to D. We cannot conclude that the metric Z improves the approach.

4.4 Discussion

4.4.1 Threats to validity

The methods we use to extract the dependencies extract only a subset of them, as explained in Sect. 4.2.1. It is possible that at runtime several additional dependencies exist, leading to more cycles. This phenomenon could change the precision or recall computed in the article. Another threat to validity is that each dependency is eval-

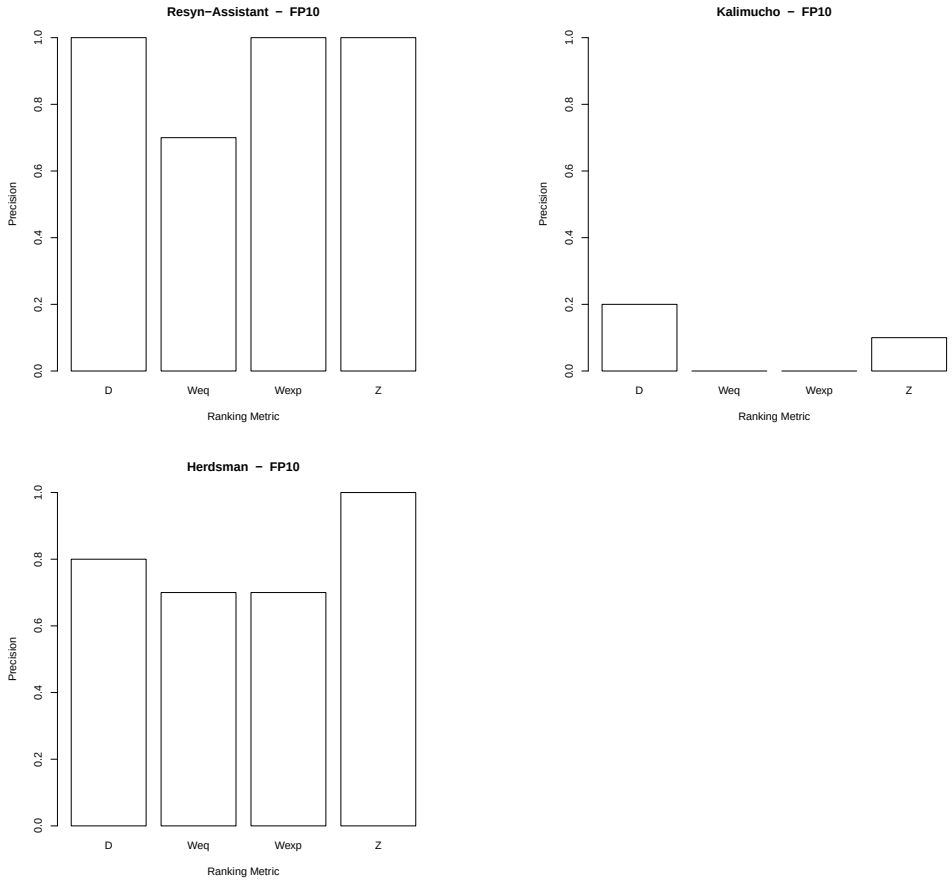

Figure 5 – The precision of FP_{10}.

uated by only one developer. The developers may have made mistakes. This threat to validity can be avoided by making evaluation by multiple developers and verifying that the evaluation is the same for each of them. Unfortunately, it was not possible in our experiment because the programs had only one main developer available. Finally, the number of studied software applications is also a threat to validity. A population of four programs cannot provide statistically significant results. Results could be different for programs with different characteristics than the ones we chose.

5 Related Work

Several tools and approaches have been introduced over the years to deal with the problem of cyclic dependencies among packages and classes in a software system. These approaches can be roughly classified as follows: 1) approaches working at the package level, 2) approaches working at the class level, 3) approaches using graph theory algorithms and 4) approaches based on dependency matrix algorithms.

As a general rule, these approaches are concerned by detecting and reporting cycles using Tarjan SCC algorithm [Tar72] or some simpler algorithms. Such approaches

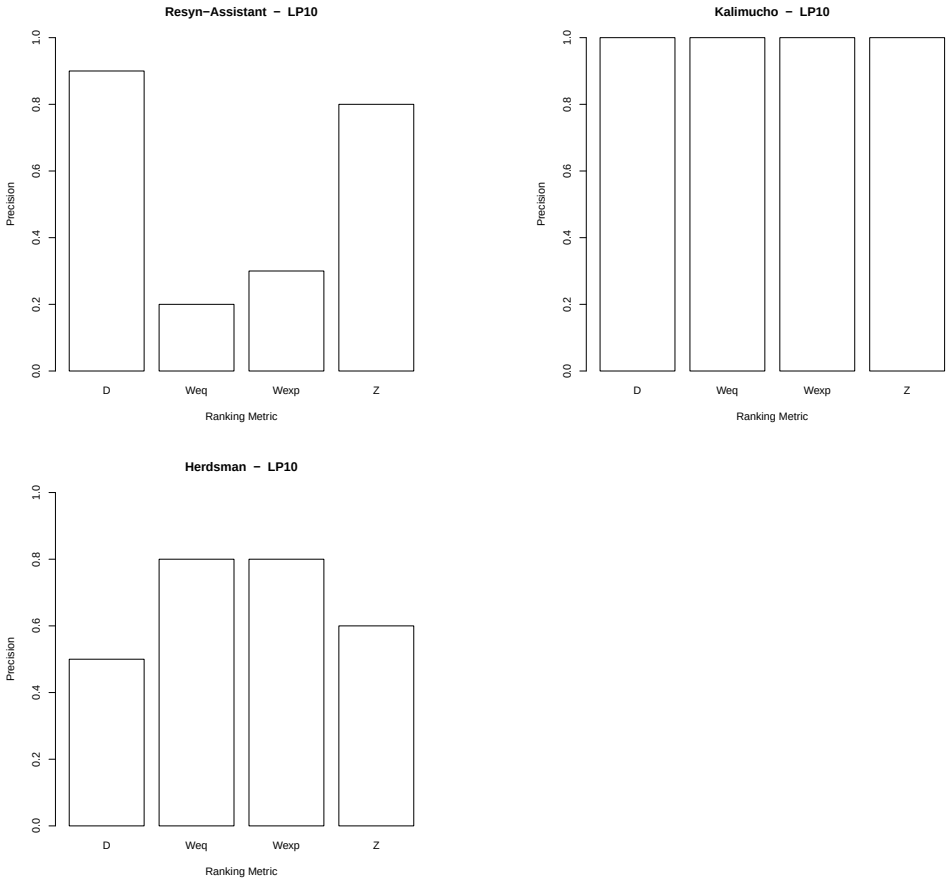

Figure 6 – The precision of LP_{10}.

do not scale to programs involving large SCCs because they do not provide a deep analysis of how such SCCs arise and how to remove cycles in a SCC. In contrast, our approach computes the information necessary to understand SCCs through subsets of elementary cycles, and is able to rank cycles by their level of undesirability.

Mudpie [Vai04] is a reporting tool to detect cyclic dependencies between packages in Smalltalk. The paper reports on a single case study performed on packages of the Refactoring Browser in Smalltalk. Classycle[11] is a reporting tool which detects SCC both at class and package level. Classycle proposes some metrics to characterize cycles but no formal definitions are proposed and their goal is unclear. Both tools rely on Tarjan SCC algorithm for detection of cycles, which make them impractical to analyze large SCCs.

PASTA [Hau02] is a tool for analyzing the dependency graph of Java packages. It focuses on detecting layers in the graph and consequently provides two heuristics to deal with cycles. One views packages in the same SCC as a single package. The other heuristic selectively ignores some dependencies until no more cycle is detected. Thus, PASTA reports on these *undesirable* dependencies which should be removed to

[11] http://classycle.sourceforge.net

break cycles. The paper reports on a case study analyzing the Java core package with effective results. It would be interesting to compare the heuristics for undesirable dependencies with our distance metric for undesired cycles.

JooJ [MT07b] is an approach to detect and remove cyclic dependencies between classes. The principle of JooJ is to find statements creating cyclic dependencies directly in the code editor, allowing the developer to solve the problem as it appears. It computes the SCC using Tarjan to detect cycles among classes. It also computes an approximation of the minimal set of edges to remove in order to make the dependency graph totally acyclic. This NP-complete problem is called minimum feedback arc set in the literature. It highlights therefore the minimum number of statements that one needs to remove to suppress all class cycles. However, no study is made to validate this approach: it is possible that the selected dependencies are in fact not to be removed. With the same idea, Ozone [LDA10] is an approach to suggest package layers to reengineers facing the presence of cyclic dependencies. This approach proposes an organization of packages (even in presence of cycles) in multiple layers by removing dependencies that are considered as undesired. This approach can be run automatically, it also supports human inputs and constraints.

Byecycle[12] is an Eclipse plugin to visualize dependencies at class level. It detects and colors in red dependencies involved in cycles. By construction, a set of red edges highlights SCC in the visualization. However, the tool does not provide further help for cycle analysis.

JDepend[13] is a tool for Java which checks Martin's principles [Mar02] for package design. In particular, it checks that the package dependency graph is acyclic. Contrary to other approaches, this tool does not detect and retrieve packages in SCCs, but simply reports for each package whether there is a cycle in its transitive dependency graph. For example, with packages A and B in cycle and package C depending upon A, JDepend reports that C depends on a cycle. It can become overwhelming if many packages depend on the same cycle (as each will report separately the cycle) yet is not exhaustive as the tool stops as soon as a cycle is detected (not reporting all cycles in the dependency graph).

Dependency Finder[14] is a set of command line tools to analyze compiled Java code with a focus on dependency graph. One tool detects cycles but at class level only. The algorithm used is not described, although it seems to report elementary cycles.

Regression and Integration Testing domain provides approaches to find the best solution to remove cycles between entities. Le Traon et al. [LTJJM00], Briand et al. [BLW01], Tai and Daniels [TD97] propose models to break Strongly Connected Components (SCC). The goal of this kind of algorithm is to minimize stub creation. Le Hanh et al. [HATJ01] propose an experimental comparison of four approaches to break SCC for stub minimization. The goal is to find the best candidate that can remove cycles to build an integration order. As our algorithm, some approaches propose the differentiation of the kind of dependencies to avoid removing inheritance relationships [KGH$^+$96, TD97].

Dependency structural matrix [SDE91] is an approach developed for process analysis. It visualizes dependencies between some elements (tasks, processes, modules) using the adjacency matrix representation. Several algorithms are defined on the dependency matrices. The main step, called *matrix partitioning*, has a similar output

[12] http://byecycle.sourceforge.net
[13] http://clarkware.com/software/JDepend.html
[14] http://depfind.sourceforge.net

to SCC in a directed graph. Dependency matrices rely on visualization to understand cycles. They make direct cycles easy to spot but indirect cycles are hard to understand with this approach. Lattix [SJSJ05] and eDSM [LDDB09] are two adaptations of dependency matrix to the visualization of package dependencies. They highlight cycles in SCC and can be used as a starting point to understand the architecture of the system. However, due to their limitations in visualizing indirect cycles, they do not benefit from our work which decomposes SCCs in direct and indirect cycles. Instead, we view our work as complementary with DSM as a high level tool and other tools for fine-grained analysis of cycles. Some other approaches propose to recover software structure and visualize the organization of classes and files [MM06]. To understand the complexity of large object-oriented software systems and particularly the package structure, there are some visualization tools [DGK06, DL05, BDL05, LSP05]. Package Blueprint [DPS+07] shows the communications between packages.

Dong and Godfrey [DG07] propose an approach to study dependencies between packages and to give a new meaning to packages with (i) characterization of external properties of components, (ii) usage of resource and (iii) connectors. It helps the maintainers to understand the nature of package dependencies.

Lungu et al. [LLG06] propose a collection of package patterns to help reengineers to understand large software system. They propose to recover architecture based on package information and an automatic process to recover defined patterns. Then they propose a user interface to interact with the package structure. This approach is useful to understand the behavior of a package in the system. It can provide information about the position of a package in a layered organization.

6 Conclusion and Future Work

In this article, we presented two contributions that assist the developers to understand and remove the cycles among packages of a large software system.

- First, we presented an efficient algorithm that decomposes a SCC. This algorithm retrieves a set of short cycles that covers all dependencies of the SCC. It has a polynomial time and space complexity.
- Second, we introduced new metrics that evaluate the level of undesirability of a cycle. These metrics are based upon the two characteristics of packages: the notion of distance between packages involved in the cycle (called diameter), and the notion of dependency weight. We showed that the diameter provides good results for ordering package cycles by undesirability.

Since our algorithm has a low complexity, it can be applied at maintenance-time as well as at development-time, preventing cycles before they become too large. We validate our approach on several case-studies on mature real-world programs in Java. It shows that our approach has a practical interest and is easy to adapt to various object languages.

We plan to work on adapting and applying our tool to legacy procedural languages like *C* or *ADA*, because we believe that cycles are frequent in legacy code. An approach able to help the developers to remove some of them would ease the maintenance effort spent on these systems.

References

[BDL05] Michael Balzer, Oliver Deussen, and Claus Lewerentz. Voronoi treemaps for the visualization of software metrics. In *SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization*, pages 165–172, New York, NY, USA, 2005. ACM. doi:10.1145/1056018.1056041.

[BLW01] L.C. Briand, Y. Labiche, and Yihong Wang. Revisiting strategies for ordering class integration testing in the presence of dependency cycles. In *Software Reliability Engineering, 2001. ISSRE 2001. Proceedings. 12th International Symposium on*, pages 287 – 296, nov. 2001. doi:10.1109/ISSRE.2001.989482.

[DG07] Xinyi Dong and M.W. Godfrey. System-level usage dependency analysis of object-oriented systems. In *ICSM 2007*. IEEE Comp. Society, 2007. doi:10.1109/ICSM.2007.4362650.

[DGK06] Stéphane Ducasse, Tudor Gîrba, and Adrian Kuhn. Distribution map. In *Proceedings of 22nd IEEE International Conference on Software Maintenance (ICSM '06)*, pages 203–212, Los Alamitos CA, 2006. IEEE Computer Society. Available from: http://scg.unibe.ch/archive/papers/Duca06cDistributionMap.pdf, doi:10.1109/ICSM.2006.22.

[DL05] Stéphane Ducasse and Michele Lanza. The Class Blueprint: Visually supporting the understanding of classes. *Transactions on Software Engineering (TSE)*, 31(1):75–90, January 2005. Available from: http://scg.unibe.ch/archive/papers/Duca05bTSEClassBlueprint.pdf, doi:10.1109/TSE.2005.14.

[DPS+07] Stéphane Ducasse, Damien Pollet, Mathieu Suen, Hani Abdeen, and Ilham Alloui. Package surface blueprints: Visually supporting the understanding of package relationships. In *ICSM '07: Proceedings of the IEEE International Conference on Software Maintenance*, pages 94–103, 2007. Available from: http://scg.unibe.ch/archive/papers/Duca07cPackageBlueprintICSM2007.pdf.

[FDL+11] Jean Rémi Falleri, Simon Denier, Jannik Laval, Philipe Vismara, and Stéphane Ducasse. Efficient retrieval and ranking of undesired package cycles in large software systems. In *Proceedings of the 49th International Conference on Objects, Models, Components, Patterns (TOOLS-Europe'11)*, Zurich, Switzerland, June 2011.

[HATJ01] Vu Le Hanh, Kamel Akif, Yves Le Traon, and Jean-Marc Jézéquel. Selecting an efficient oo integration testing strategy: An experimental comparison of actual strategies. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 381–401, London, UK, UK, 2001. Springer-Verlag. Available from: http://portal.acm.org/citation.cfm?id=646158.679879.

[Hau02] Edwin Hautus. Improving java software through package structure analysis. In *IASTED International Conference Software Engineering and Applications*, 2002.

[KGH+96] David C. Kung, Jerry Gao, Pei Hsia, Yasufumi Toyoshima, and Cris Chen. On regression testing of object-oriented programs. *J. Syst.*

Softw., 32:21–40, January 1996. Available from: http://portal.acm.org/citation.cfm?id=218153.218155, doi:10.1016/0164-1212(95)00047-X.

[LDA10] Jannik Laval, Stéphane Ducasse, and Nicolas Anquetil. Ozone: Package layered structure identification in presence of cycles. In *9th BElgian-NEtherlands software eVOLution seminar (BENEVOL 2010)*, Lille, France, 2010.

[LDDB09] Jannik Laval, Simon Denier, Stéphane Ducasse, and Alexandre Bergel. Identifying cycle causes with enriched dependency structural matrix. In *WCRE '09: Proceedings of the 2009 16th Working Conference on Reverse Engineering*, Lille, France, 2009.

[LLG06] Mircea Lungu, Michele Lanza, and Tudor Gîrba. Package patterns for visual architecture recovery. In *Proceedings of CSMR 2006 (10th European Conference on Software Maintenance and Reengineering)*, pages 185–196, Los Alamitos CA, 2006. IEEE Computer Society Press. Available from: http://scg.unibe.ch/archive/papers/Lung06aPackagePatterns.pdf, doi:10.1109/CSMR.2006.39.

[LSP05] Guillaume Langelier, Houari A. Sahraoui, and Pierre Poulin. Visualization-based analysis of quality for large-scale software systems. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 214–223, New York, NY, USA, 2005. ACM. Available from: http://dx.doi.org/10.1145/1101908.1101941, doi:10.1145/1101908.1101941.

[LTJJM00] Y. Le Traon, T. Jeron, J.-M. Jezequel, and P. Morel. Efficient object-oriented integration and regression testing. *Reliability, IEEE Transactions on*, 49(1):12 –25, March 2000. doi:10.1109/24.855533.

[Mar02] Robert Cecil Martin. *Agile Software Development. Principles, Patterns, and Practices*. Prentice-Hall, 2002.

[MM06] Brian S. Mitchell and Spiros Mancoridis. On the automatic modularization of software systems using the bunch tool. *IEEE Transactions on Software Engineering*, 32(3):193–208, 2006.

[MT07a] Hayden Melton and Ewan Tempero. An empirical study of cycles among classes in java. *Empirical Software Engineering*, 12(4):389–415, 2007. doi:10.1007/s10664-006-9033-1.

[MT07b] Hayden Melton and Ewan D. Tempero. Jooj: Real-time support for avoiding cyclic dependencies. In *14th Asia-Pacific Software Engineering Conference*, pages 87–95. IEEE Computer Society, 2007.

[Par78] David Lorge Parnas. Designing software for ease of extension and contraction. In *International Conference on Software Engineering (ICSE'78)*, pages 264–277, 1978.

[SDE91] David A. Gebala Steven D. Eppinger. Methods for analyzing design procedures. In *ASME Conference on Design Theory and Methodology*, pages 227–233, 1991. Miami.

[SJSJ05] Neeraj Sangal, Ev Jordan, Vineet Sinha, and Daniel Jackson. Using dependency models to manage complex software architecture. In *Proceedings of OOPSLA'05*, pages 167–176, 2005.

[Tar72] Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[Tar73] Robert Endre Tarjan. Enumeration of the elementary circuits of a directed graph. *SIAM J. Comput.*, 2(3):211–216, 1973.

[TD97] Kuo-Chung Tai and F.J. Daniels. Test order for inter-class integration testing of object-oriented software. In *Computer Software and Applications Conference, 1997. COMPSAC '97. Proceedings., The Twenty-First Annual International*, pages 602 –607, aug 1997. doi:10.1109/CMPSAC.1997.625079.

[TSWW11] Craig Taube-Schock, Robert Walker, and Ian Witten. Can we avoid high coupling? In Mira Mezini, editor, *ECOOP 2011 – Object-Oriented Programming*, volume 6813 of *Lecture Notes in Computer Science*, pages 204–228. Springer Berlin / Heidelberg, 2011.

[Vai04] Daniel Vainsencher. Mudpie: layers in the ball of mud. *Computer Languages, Systems & Structures*, 30(1-2):5–19, 2004.